Chapter 5

To C or Not to C

In This Chapter

- Inserting notes for yourself
- Using fancy commenting techniques
- Borrowing C++ comments
- Disabling code with comments
- Avoiding nested comments

An important part of programming is remembering what the heck it is you're doing. I'm not talking about the programming itself — that's easy to remember, and you can buy books and references galore in case you don't. Instead, the thing you have to remember is what you are attempting to make a program do at a specific spot. You do that by inserting a *comment* in your source code.

Comments aren't really necessary for the small programs you're doing in this book. Comments don't begin to become necessary until you write larger programs — on the scope of Excel or Photoshop — where you can easily lose your train of thought. To remind yourself of what you're doing, you should stick a comment in the source code, explaining your approach. That way, when you look at the source code again, your eyes don't glaze over and the drool doesn't pour, because the comments remind you of what's going on.

Adding Comments

Comments in a C program have a starting point and an ending point. Everything between those two points is ignored by the compiler, meaning that you can stick any text in there — anything — and it doesn't affect how the program runs.

```
/* This is how a comment looks in the C language */
```

This line is a fine example of a comment. What follows is another example of a comment, but the type that gives this book its reputation:

```
/*
Hello compiler!  Hey, error on this: pirntf!
Ha! Ha! You can't see me!  Pbbtbtbt!
Nya! Nya! Nya!
*/
```

- The beginning of the comment is marked by the slash and the asterisk: `/*`.
- The end of the comment is marked by the asterisk and the slash: `*/`.
- Yup, they're different.
- The comment is not a C language statement. You do not need a semicolon after the `*/`.

A big, hairy program with comments

The following source code is MADLIB1.C. It uses the `printf()` and `scanf()` functions described in Chapter 4 to create a short yet interesting story:

```
/*
MADLIB1.C Source Code
Written by (your name here)
*/

#include <stdio.h>

int main()
{
    char adjective[20];
    char food[20];
    char chore[20];
    char furniture[20];

/* Get the words to use in the madlib */

    printf("Enter an adjective:");          /* prompt */
    scanf("%s",&adjective);                  /* input */
    printf("Enter a food:");
    scanf("%s",&food);
    printf("Enter a household chore (past tense):");
    scanf("%s",&chore);
    printf("Enter an item of furniture:");
    scanf("%s",&furniture);

/* Display the output */
```

```
    printf("\n\nDon't touch that %s %s!\n",adjective,food);
    printf("I just %s the %s!\n",chore,furniture);

    return(0);
}
```

Type the source code exactly as written. The only thing new should be the comments. Each one begins with /* and ends with */. Make sure that you get those right: A slash-asterisk begins the comment, and an asterisk-slash ends it. (If you're using a color-coded editor, you see the comments all coded in the same color.)

Save the file to disk and name it MADLIB1.C.

Compile. Run.

Here is a sample of the program's output:

```
Enter an adjective:hairy
Enter a food:waffle
Enter a household chore (past tense):vacuumed
Enter an item of furniture:couch

Don't touch that hairy waffle!
I just vacuumed the couch!
```

Oh, ha-ha! Ouch! My sides!

- This program is long and looks complex, but it doesn't use any new tricks. Everything here, you have seen already: `char` to create string variables, `printf()` to display text and string variables, and `scanf()` to read the keyboard. Yawn.
- MADLIB1.C uses these four string variables: `adjective`, `food`, `chore`, and `furniture`. All four are created by the `char` keyword, and 20 characters of storage are set aside for each one. Each of the string variables is filled by `scanf()` with your keyboard input.
- Each of the final `printf()` functions contains two `%s` placeholders. Two string variables in each function supply the text for the `%s` placeholders.
- The second-to-last `printf()` function begins with two newline characters, `\n \n`. These characters separate the input section, where you enter the bits of text, from the program's output. Yes, newlines can appear anywhere in a string, not just at the end.
- MADLIB1.C has five comments. Make sure that you can find each one. Notice that they're not all the same, yet each one begins with /* and ends with */.

Why are comments necessary?

Comments aren't necessary for the C compiler. It ignores them. Instead, comments are for you, the programmer. They offer bits of advice, suggestions for what you're trying to do, or hints on how the program works. You can put anything in the comments, though the more useful the information, the better it helps you later on.

Most C programs begin with a few lines of comments. All my C programs start with information such as the following:

```
/* COOKIES.C
Dan Gookin, 1/20/05 @ 2:45 a.m.
Scan Internet cookie files for expired
dates and delete.
*/
```

These lines tell me what the program is about and when I started working on it.

In the source code itself, you can use comments as notes to yourself, such as

```
/* Find out why this doesn't work */
```

or this:

```
save=itemv;    /* Save old value here */
```

or even reminders to yourself in the future:

```
/*
Someday you will write the code here that makes
the computer remember what it did last time this
program ran.
*/
```

The point is that comments are notes *for yourself.* If you were studying C programming in school, you would write the comments to satiate the fixations of your professor. If you work on a large programming project, the comments placate your team leader. For programs you write, the comments are for you.

Comment Styles of the Nerdy and Not-Quite-Yet-Nerdy

The MADLIB1.C program contains five comments and uses three different commenting styles. Though you can comment your programs in many more ways, these are the most common:

```
/*
MADLIB1.C Source Code
Written by Mike Rowsoft
*/
```

Ever popular is the multiline approach, as just shown. The first line starts the comment with the /* all by itself. Lines following it are all comments, remarks, or such and are ignored by the compiler. The final line ends the comment with */ all by itself. Remember that final /*; otherwise, the C compiler thinks that your whole program is just one big, long comment (possible, but not recommended).

```
/* Get the words to use in the madlib */
```

This line is a single-line comment, not to be confused with a C language statement. The comment begins with /* and ends with */ all on the same line. It's 100 percent okey-dokey, and, because it's not a statement, you don't need a semicolon.

Finally, you can add the "end of line" comment:

```
    printf("Enter an adjective:");       /* prompt */
```

After the preceding `printf` statement plus a few taps of the Tab key, the /* starts a comment, and */ ends it on the same line.

Bizarr-o comments

During my travels, I have seen many attempts to make comments in C programs look interesting. Here's an example:

```
/*****************************************
**  Commander Zero's Excellent Program  **
*****************************************/
```

This comment works. It contains lots of asterisks, but they're all still stuck between /* and */, making it a viable comment.

I have used this before in my programs:

```
/*
 * This is a long-winded introduction to an
 * obscure program written by someone at a
 * university who's really big on himself and
 * thinks no mere mortal can learn C -- and who
 * has written three "C" books to prove it.
 */
```

The idea in this example is to create a “wall of asterisks” between the /* and */, making the comment stick out on the page.

Another example I often use is this:

```
/*****************************************/
```

That line of asterisks doesn’t say anything, yet it helps to break up different sections in a program. For example, I may put a line of asterisks between different functions so that I can easily find them.

The bottom line: No matter what you put between them, a comment must start with /* and end with */.

C++ comments

Because today’s C compilers also create C++ code, you can take advantage of the comment style used by C++ in your plain old C programs. I mention it simply because the C++ comment style can be useful, and it’s permitted if you want to borrow it.

In C++, comments can start with a double slash, //. That indicates that the rest of the text on the line is a comment. The end of the line marks the end of the comment:

```
//This is another style of comment,
//one used in C++
```

This commenting style has the advantage that you don’t have to both begin and end a comment, making it ideal for placing comments at the end of a C language statement, as shown in this example:

```
    printf("Enter an adjective:");           // prompt
    scanf("%s",&adjective);                  // input
```

These modifications to the MADLIB1.C program still keep the comments intact. This method is preferred because it’s quick; however, /* and */ have the advantage of being able to rope in a larger portion of text without typing // all over the place.

Using Comments to Disable

Comments are ignored by the compiler. No matter what lies between the /* and the */, it's skipped over. Even vital, lifesaving information, mass sums of cash, or the key to eternal youth — all these are ignored if they're nestled in a C language comment.

Modify the MADLIB1.C source code, changing the last part of the program to read:

```
/* Display the output */

/*
    printf("\n\nDon't touch that %s %s!\n",adjective,food);
    printf("I just %s the %s!\n",chore,furniture);
*/
```

To make the modification, follow these cinchy steps:

1. Insert a line with /* on it before the first printf() function in this example.
2. Insert a line with */ on it after the second printf() function.

With the last two printf() statements disabled, save the file to disk and recompile it. It runs as before, but the resulting "mad lib" isn't displayed. The reason is that the final two printf() functions are "commented out."

- You can use comments to disable certain parts of your program. If something isn't working correctly, for example, you can "comment it out." You may also want to include a note to yourself, explaining why that section is commented out.
- Sometimes you may notice that something which should be working isn't working. The reason is that you may have *accidentally* commented it out. Always check your /* and */ comment bookends to make sure that they match up the way you want them to.
- By using an editor with color-coded text, you can easily spot missing */ characters to end a comment. If you notice that a greater chunk of your source code is colored as a comment, a misplaced */ is probably to blame.

TIP

The Perils of "Nested" Comments

The most major of the faux pas you can commit with comments is to "nest" them, or to include one set of comments inside another. To wit, I present the following C program fragment:

```
if(all_else_fails)
   {
       display_error(erno);        /* erno is already set */
       walk_away();
   }
else
   get_mad();
```

Don't worry about understanding this example; it all comes clear to you later in this book. However, notice that the `display_error` function has a comment after it: `erno is already set`. But suppose that, in your advanced understanding of C that is yet to come, you want to change the gist of this part of the program so that only the `get_mad()` function is executed. You comment out everything except that line to get it to work:

```
/*
if(all_else_fails)
   {
   display_error(erno);     /* erno is already set */
   walk_away();
   }
else
*/
   get_mad();
```

Here, the C compiler sees only the `get_mad` function, right?

Wrong! The comment begins on the first line with the `/*`. But it ends on the line with the `display_error()` function. Because that line ends with `*/` — the comment bookend — that's the end of the "comment." The C compiler then starts again with the `walk_away` function and generates a parse error on the rogue curly brace floating in space. The second comment bookend (just above the `get_mad()` function) also produces an error. Two errors! How heinous.

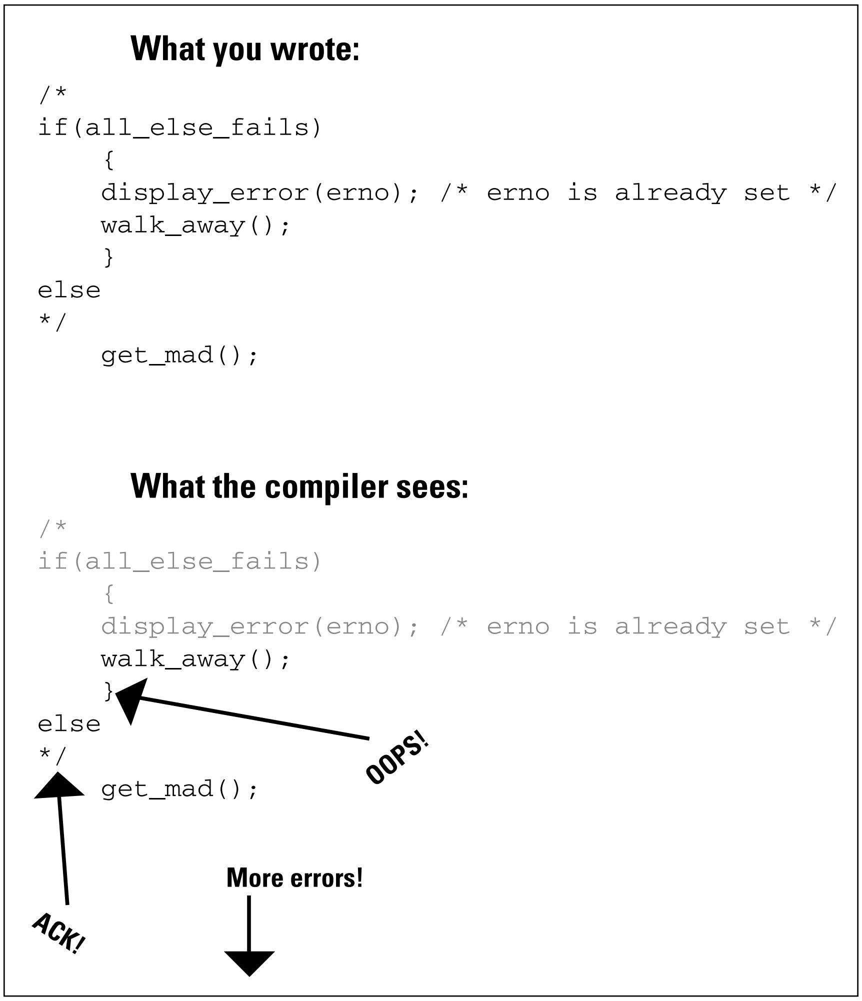
This example shows a *nested comment,* or a comment within a comment. It just doesn't work. Figure 5-1 illustrates how the C compiler interprets the nested comment.

To avoid the nested-comment trap, you have to be careful when you're disabling portions of your C program. The solution in this case is to uncomment the `erno is already set` comment. Or, you can comment out each line individually, in which case that line would look like this:

```
/*  display_error(erno);    /* erno is already set */
```

This method works because the comment still ends with `*/`. The extra `/*` inside the comment is safely ignored.

- Yeah, nested comments are nasty, but nothing you need to worry about at this point in the game.
- Note that the C++ style of comments, `//`, doesn't have a nesting problem.



```
/*
if(all_else_fails)
    {
    display_error(erno); /* erno is already set */
    walk_away();
    }
else
*/
    get_mad();
```

```
/*
if(all_else_fails)
    {
    display_error(erno); /* erno is already set */
    walk_away();
    }
else
*/
    get_mad();
```

Figure 5-1: The perils of a nested comment.